Using Model Checking to Validate AI Planner Domain Models

John Penix, Charles Pecheur and Klaus Havelund
Automated Software Engineering Group
NASA Ames Research Center
M/S 269-3
Moffett Field, CA 94035
jpenix,pecheur,havelund@ptolemy.arc.nasa.gov

Abstract

This report describes an investigation into using model checking to assist validation of domain models for the HSTS planner. The planner models are specified using a qualitative temporal interval logic with quantitative duration constraints. We conducted several experiments to translate the domain modeling language into the SMV, Spin and Murphi model checkers. This allowed a direct comparison of how the different systems would support specific types of validation tasks. The preliminary results indicate that model checking is useful for finding faults in models that may not be easily identified by generating test plans.

1 Introduction

In the classical approach to analyzing the correctness of a piece of software is broken into two tasks: verification and validation. Verification is the task of making sure that the software implementation complies with a stated set of requirements. Validation is making sure that the stated requirements correctly reflect the needs of the end user.

In the realm of artificial intelligence and knowledge-based systems, the role of validation shifts slightly. These systems are generally divided into two parts: a knowledge base that is used as a model of the environment with which the program interacts, and a reasoning algorithm that manipulates the knowledge base during program execution. The accuracy of the knowledge base with respect to the real environment has direct implications on the performance of the AI system. A system with a totally correct reasoning algorithm will be ineffective if its model of the world is flawed. Therefore, validation becomes a critical task of evaluating a part of the system to be deployed.

We are currently investigating the use of model checking [2, 4, 5] to assist in the validation of models used in the HSTS Planner [6], a model-based planning system that is being used in the Remote Agent autonomous control system architecture [7]. The planner takes an initial state and a goal as inputs and produces a plan for achieving that goal. The planning algorithm guarantees that the generated plan is consistent with a set of temporal constraints that model physical limitations of the controlled system and the environment.

Because domain models are often composed of a large number of tightly coupled constraints, the interactions among constraints can become conceptually unmanageable very quickly. Therefore, it is desirable to have methods to validate the model by finding inconsistencies in the model and determining whether implicit properties of the model can be derived from the set of explicit constraints.

2 Constraint Model Specifications

The HSTS Planner domain models are described using the HSTS Domain Description Language (DDL), an object-oriented constraint specification language based on Allen's temporal interval logic [1].

DDL models define object classes and instances. Each object encapsulates a number of state variables. A state variable is declared to be *controllable* if it can be modified during scheduling and *uncontrollable* otherwise. For example, a robot object class could be defined and instantiated as follows:

```
(Define_Object_Class Robot
  :state_variables
  ((Controllable Task)
   (Controllable Location)))

(Define_Object Robot Robbie)
```

The language provides special support for the definition of objects without any properties as sets of labels:

```
(Define_Label_Set Room (Kitchen Hallway LivingRoom))
```

A set of *member values* is defined for each state variable. The member values are called *predicates* and may have parameters. The name of a predicate and the value of its parameters define the value of a state.

Predicate parameters can be objects, labels or built in types. For example, possible values for the Task and Location state variables in the Robot class are defined in Figure 1. In this example, the Task state variable for the Robot class can either has the value `Moving(`x`,`y`)`, where x and y are locations, or it has the value `Idle`. The Location state in the Robot class can have the value `In_Room(`x`)`, where x is a location.

```
(Define_Predicate Moving
  ((Room From)
   (Room To)))

(Define_Predicate Idle)
(Define_Member_Values ((Robot Task))
  (Moving Idle)

(Define_Predicate In_Room
  ((Room R)))
(Define_Member_Values ((Robot Location))
  (In_Room))
```

Figure 1: Example Predicate and Member Value Definitions

Temporal Relation	**Inverse Relation**	**Endpoint Relation**
T1 **before** (d D) T2	T2 **after** (d D) T1	d $\leq$ T2.start - T1.end $\leq$ D
T1 **starts_before** (d D) T2	T2 **starts_after** (d D) T1	d $\leq$ T2.start - T1.start $\leq$ D
T1 **ends_before** (d D) T2	T2 **ends_after** (d D) T1	d $\leq$ T2.end - T1.end $\leq$ D
T1 **starts_before_end** (d D) T2	T2 **ends_after_start** (d D) T1	d $\leq$ T2.end - T1.start $\leq$ D
T1 **contains** ((a A) (b B)) T2	T2 **contained_by** ((a A) (b B)) T1	a $\leq$ T2.start - T1.start $\leq$ A b $\leq$ T1.end - T2.end $\leq$ B
T1 **parallels** ((a A) (b B)) T2	T2 **paralleled_by** ((a A) (b B)) T1	a $\leq$ T2.start - T1.start $\leq$ A b $\leq$ T2.end - T1.end $\leq$ B

Table 1: HSTS DLL Temporal Relations

A DDL model is constrained by defining compatibility conditions that must hold between different values of state variables. The constraints are specified in terms of temporal intervals (called *tokens*) during which a sate variable holds a specific value or set of values. The compatible relationships between tokens are specified using a set of predefined temporal operators. The operators permit the expression of all possible temporal relationships between the start and end points of two tokens. The DDL temporal relationships are shown in Table 1. DDL also supports abbreviations for common temporal relations as shown in Table 2.

DDL compatibility specifications are defined using *token descriptors*. A token descriptor is a pattern that describes attributes of a set of tokens. These patterns can refer to either one (SINGLE) or a sequence of tokens (MULTIPLE). Compatibility specifications have the form:

(*master-token-descriptor compatibility-tree*)

Abbreviation	**Temporal Relation**
before	**before** $(0 +\infty)$
after	**after** $(0 +\infty)$
meets	**before** (0 0)
met_by	**after** (0 0)
starts	**starts_before** (0 0) = starts_after (0 0)
ends	**ends_before** (0 0) = ends_after (0 0)
contains	**contains** $((0 +\infty)\ (0 +\infty))$
contained_by	**contained_by** $((0 +\infty)\ (0 +\infty))$
equal	**contains** ((0 0) (0 0))

Table 2: Abbreviations for Common Temporal Relations

where *compatibility-tree* is an and/or tree of pairs of temporal relations and token descriptors. For example, we can specify that the robot can only move from the Kitchen to the LivingRoom by going through the Hallway as follows:

```
(Define_Compatibility
 (SINGLE ((Robot Task)) ((Moving (Kitchen LivingRoom))))
 :compatibility_spec
 (AND (met_by (SINGLE ((Robot Location)) ((In_Room(Kitchen)))))
      (equals (SINGLE ((Robot Location)) ((In_Room(Hallway)))))
      (meets (SINGLE ((Robot Location)) ((In_Room(LivingRoom)))))))
```

In addition, variables can be used to constrain parameter values between the master token descriptor and the token descriptors in the compatibility tree.

3 Model Checking for DDL

For the purpose of translating HSTS models into suitable inputs for model checkers, we need to express HSTS models in terms of states and transitions between states. In HSTS, the state of the system is defined by the values of all state variables of all objects. A transition occurs when the value of at least one state variable changes.

For describing what happens upon a transition, we take the convention that V (resp. V') denotes the value of variable V before (resp. after) the transition. We then define the following abbreviations for convenience:

$$\texttt{(SV to P)} \equiv \texttt{(SV != P) AND (SV' == P)}$$
$$\texttt{(SV from P)} \equiv \texttt{(SV == P) AND (SV' != P)}$$

DDL Constraint	**State Translation Relation**
`((SINGLE (SV1 P1)) meets (SINGLE (SV2 P2)))`	`(SV1 from P1)` $\Rightarrow$ `(SV2 to P2)`
`((SINGLE (SV1 P1)) met_by (SINGLE (SV2 P2)))`	`(SV1 to P1)` $\Rightarrow$ `(SV2 from P2)`
`((SINGLE (SV1 P1)) starts (SINGLE (SV2 P2)))`	`(SV1 to P1)` $\Rightarrow$ `(SV2 to P2)`
`((SINGLE (SV1 P1)) ends (SINGLE (SV2 P2)))`	`(SV1 from P1)` $\Rightarrow$ `(SV2 from P2)`

`((SINGLE (SV1 P1)) contained_by (SINGLE (SV2 P2)))`	`((SV1 == P1)` $\Rightarrow$ `(SV2 == P2))`
	`AND ((SV1' == P1)` $\Rightarrow$ `(SV2' == P2))`

`((SINGLE (SV1 P1)) equals (SINGLE (SV2 P2)))`	`(SV1 to P1)` $\Rightarrow$ `(SV2 to P2)`
	`AND (SV1 from P1)` $\Rightarrow$ `(SV2 from P2)`
	`AND ((SV1 == P1)` $\Rightarrow$ `(SV2 == P2))`
	`AND ((SV1' == P1)` $\Rightarrow$ `(SV2' == P2))`

Table 3: Translation from DDL Constraints to State Transition Computation Model

Using these, some of the abbreviated temporal relations from Table 2 can be expressed as a logical relation that describes legal transitions between states. Table 3 shows the translation for six of the basic DDL qualitative relations.

Some of the translations can be further simplified. For example, the `contained_by` translation can be simplified to

`((SV1' == P1)` $\Rightarrow$ `(SV2' == P2))`

plus checking that in the initial state

`((SV1 == P1)` $\Rightarrow$ `(SV2 == P2))`

This simplification was used in the SMV model.

This generic translation was used as the basis for translation into the input languages for 3 different model checkers: Spin [5], SMV [2] and Murphi [4]. The translation had to be done slightly differently for each due to differences in their input languages. For example, the translation into both Spin and Murphi includes a small program that selects the next state of the model based on the state transition relation. In SMV, the input languages allows the specification of a state transition relation as a propositional formula, simplifying the translation.

4 Results

We tested our translation on a small model of an autonomous robot. The model consisted of a battery powered robot that could move between three locations: A, B, and In_Between. The goal of the robot was to fix a hole at location A. The model had 65 temporal constraints which allowed 16320 reachable states out of a potential 559872 states. The constrains in the model dictated such things as:

```
 ((Robot.Task = Moving) contained_by (Hole.Charge = Charge_Full))
   ((Robot.Task = Fixing_Hole) meets (Hole.Status = Hole_Fixed))
```

The analysis identified several potential flaws in the model that were not identified during testing.

4.1 Expressibility

The subset of the language that is covered by the initial translation is expressive enough to cover the majority of the modeling done for the robot. We were not able to capture a couple of quantitative durations that were used in the model. The inability to translate nonlocal constraints was not a limitation in this example.

4.2 Analysis Capabilities

We analyzed the model for both safety and liveness properties. Safety properties state that nothing wrong ever happens, e.g. that no deadlock occurs. They are simple to verify: the model checker checks if any reachable state violates the property. Liveness properties state that something good must eventually happens, e.g. that a query is always answered. They are more complex to handle because they apply to runs rather than states. Some model checkers, like Murphi, do not support liveness properties; others, such as SPIN and SMV, use a more elaborate state space exploration algorithm to verify them.

4.2.1 Safety Properties

Safety properties can be used to check whether plans exist within a domain model. This is done by checking whether there is a path that leads to a goal state, or, cast as a safety property, checking if it is never the case that the system reaches a goal state. If there is such a case, the model checker will return a series of state transitions that will lead to the goal state, which is in essence a plan.

For example, in SMV we could ask if there was a plan where the robot fixes the hole using the query

```
!EF (Hole.Status = Hole_Fixed)
```

which reads, "for all initial states, there does not **E**xist a **F**uture state where the hole is fixed." In an early version of the model, this analysis yields "true", meaning that the hole could never be fixed. Further queries revealed the fact that the robot could not move. This was because of an overly general constraint that was intended to constrain movement to and from locations A and B, but also ended up constraining location In_Between. The fix was to replace the general constraint with individual constraints for A and B. After the fix, analysis of the above specification gave a sequence of states that lead to the hole being fixed.

There were several other queries that were useful for inspecting the model. First, it was interesting to see if a plan existed from *any* initial state in the model, not just *some* initial state as is the case above. This can be done with the query

```
EF (Hole.Status = Hole_Fixed)
```

which reads, "for all initial states, there **E**xists a **F**uture state where the hole is fixed." Note that this specification is not the logical opposite of the previous query, because SMV includes an implicit quantification over all legal initial states, requiring "for all initial states" to be read before any query. Analysis of the second specification gives an example of a initial state from which the hole cannot be fixed, i.e. one where the robot starts away from a recharge station, with no charge. This analysis therefore useful for identifying initial states that may have been unintended.

4.2.2 Liveness Properties

In the robot model, a desirable liveness property is that there *always* exists a sequence of states that leads to the hole being fixed. This can be tested using the query

```
AG EF (Hole.Status = Hole_Fixed)
```

which asks, "for all initial states, and for all states (**G**lobally) on **A**ll paths, there **E**xists a path with a **F**uture state where the hole is fixed." Analysis of this specification revealed that it was not true. An error trace was reported in which the robot fixes the hole and then the hole reappears. This leaves us in the situation where the robot is uncharged away from a recharge station and cannot fix the new hole. If the model is constrained such that a fixed hole stays fixed, then this specification becomes true.

Another liveness property is that the hole eventually gets fixed. That is, "**A**ll paths lead to a **F**uture state where the hole is fixed. In SMV, we can write this as:

```
AF (Hole.Status = Hole_Fixed)
```

Without further hypotheses, it turns out that this formula does not hold. Indeed, SMV reports a diagnostic trace where the robot remains idle forever, which is a valid behavior according to the model but unlikely to be chosen by the more proactive HSTS planner.

To obtain a more meaningful result, we consider only "fair" runs, such that no enabled move is ignored indefinitely. The property that all such fair runs eventually reach a property p is captured by the SMV formula `!E[!p U (AG !p)]` [8], so the formula above becomes:

```
!E[!(Hole.Status = Hole_Fixed) U AG !(Hole.Status = Hole_Fixed)]
```

which means that, "from all initial states, there does not **E**xist a path where the hole remains not fixed **U**ntil a point where the hole is never (**AG!**) fixed." SMV indeed reports this to be a valid property of our model.[1]

[1]SMV also has a FAIRNESS declaration for declaring application-specific fairness constraints explicitly, which has not been used here.

4.3 Performance

For the initial subset of the language addressed, the SMV model checker is much faster than its competitors (0.05 seconds vs. 30 seconds). There were two reasons for this result. The main reason is that the reachable traces through the system were shallow relative to the size of the state space. This type of model is more amiable to the symbolic state space representation and searching technique employed by SMV. This method represents sets of states as logical predicates avoiding the need to represent each state explicitly. The second reason is that SMV allows explicit specification of a state transition relation as opposed to the standard guarded command language or programming language format for model checkers. This allowed the translated DDL constraints to be mapped directly into the model checking language without the need for additional mechanisms to select the next state based on the constraints. Further studies will investigate whether SMV can maintain this advantage when the translation is expanded to handle non-local constraints and quantitative intervals.

5 Related Work

The link between planning and model checking is not new; Cimatti et al [3] use the SMV model checker as a reasoning engine to do planning. There work has the same spirit as ours because it also requires translation from a planning language to a model checking language. However, the differences between the two planning languages are significant, making the technical details dissimilar.

6 Conclusion

Our initial investigations show that model checking can be effective at finding flaws in HSTS domain models. The exhaustive search capabilities of model checking will discover modeling errors that may be overlooked by the heuristic search done during planning. Our future work will involve expanding the expressiveness of the language covered by the translation and, after determining the best target, automating the translation so that model checking can be used directly by domain experts.

Acknowledgements

We would like to thank Nicola Muscettola and Kanna Rajan for their help in understanding DDL and the HSTS planner, Lina Khatib for allowing us to use her robot model, and Nikolaj Bjorner for his enlightening discussions about the performance of various model checkers on the robot example.

References

[1] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23:123–154, 1984.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10e20 states and beyond. In *Proceedings of The International Conference on Logic in Computer Science*, 1990.

[3] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In S. Steel and R. Alami, editors, *Proceeding of the Fourth European Conference on Planning*, number 1348 in Lecture Notes in Artificial Intelligence, pages 130–142, Toulouse, France, September 1997. Springer-Verlag. Also ITC-IRST Technical Report 9705-02, ITC-IRST Trento, Italy.

[4] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[5] Gerard J. Holzmann. *Design and Verification of Protocols.* Prentice Hall, 1990.

[6] Nicola Muscettola. Hsts: Integrating planning and scheduling. In M. Zweben and M. S. Fox, editors, *Intelligent Scheduling.* Morgan Kaufmann, 1994.

[7] Barney Pell, Erann Gat, Ron Keesing, Nicola Muscettola, and Ben Smith. Plan execution for autonomous spacecraft. In *Proceedings of the 1997 International Joint Conference on Artificial Intelligence*, 1997.

[8] J. P. Queille and Joseph Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.